Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 159 – 168

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Elastic Memory Management of Virtualized Infrastructures for Applications with Dynamic Memory Requirements

Germán Moltó*, Miguel Caballer, Eloy Romero, Carlos de Alfonso

[a]*Instituto de Instrumentación para Imagen Molecular (I3M). Centro mixto CSIC Universitat Politècnica de València CIEMAT, camino de Vera s/n, 46022 Valencia, España*

Abstract

This paper addresses the impact of vertical elasticity for applications with dynamic memory requirements when running on a virtualized environment. Vertical elasticity is the ability to scale up and scale down the capabilities of a Virtual Machine (VM). In particular, we focus on dynamic memory management to automatically fit at runtime the underlying computing infrastructure to the application, thus adapting the memory size of the VM to the memory consumption pattern of the application. An architecture is described, together with a proof-of-concept implementation, that dynamically adapts the memory size of the VM to prevent thrashing while reducing the excess of unused VM memory. For the test case, a synthetic benchmark is employed that reproduces different memory consumption patterns that arise on real scientific applications. The results show that vertical elasticity, in the shape of dynamic memory management, enables to mitigate memory overprovisioning with controlled application performance penalty.

Keywords: Cloud computing, Cluster computing, Virtualization, Elasticity

1. Introduction

With the advent of virtualization, the wide use of commodity hardware and the advances in networks, the idea of utility computing, concerning the access to computing (and storage) resources on a pay-per-use basis, is taking shape. Cloud computing provides, at the moment, the closest implementation of utility computing, by providing a model for enabling ubiquitous, on-demand network access to a pool of configurable computing resources that can be rapidly provisioned and released with minimal provider interaction, according to the NIST definition [1].

Elasticity, i.e., the ability to rapidly provision and release resources, is often highlighted as one of the key features of Cloud computing [2], for it allows to dynamically adapt the underlying virtual computing infrastructure to the dynamic execution requirements of applications. This is specially true in the case of Infrastructure as a Service (IaaS) Cloud providers, where users request and release specific resources (mainly computational and storage capabilities) and pay for its usage.

On the one hand, horizontal elasticity has the ability of rapidly provisioning and releasing nodes in order to deal with an important change in the workload and to avoid additional costs (for example, from paying for unused resources). A typical example that uses horizontal elasticity is a web-based application with a fleet of n Virtual Machines (VMs) where incoming requests are handled by a load balancer that distributes them to those n VMs.

*Corresponding author. Tel.: +34963877007 Ext. 88254 ; fax: +34963877274.
E-mail address: gmolto@dsic.upv.es.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.179

Whenever the (CPU) load of VMs exceeds a certain threshold, the fleet is increased to m VMs (where $m > n$). If the load decreases, some VMs are shut down in order to reduce costs. On the other hand, vertical elasticity has the ability to rapidly modify the capabilities of single VMs, typically in terms of CPU and RAM [3].

In the last years, many research efforts in the area of elasticity in the Cloud have focused on horizontal elasticity, while few works currently address vertical elasticity. In fact a report from the European Commission on the Future of Cloud Computing states that vertical elasticity is one of the areas not fully addressed by current commercial efforts, although it is acknowledged its importance for the efficient adaptation of infrastructures to applications [4].

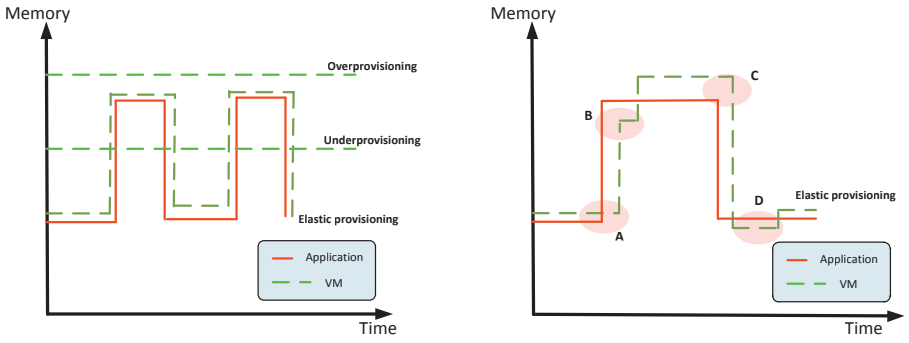
As an example, the ability to dynamically modify the memory of a VM at runtime without any service disruption represents an important capability for applications with dynamic memory requirements [5]. This means to automatically adapt the underlying virtual computing platform (i.e., the VM or the set of VMs) to the runtime profile of memory consumption of the application. This introduces a benefit for resource providers, because a reduction of the memory size of the VM increases the available memory at the host on which the VM is running. This free memory could be dedicated to other concurrent VMs being run on the same physical machine. This could also lead to reduced costs for the user if public Cloud providers offered support to these techniques (which is not the case as of 2013, considering the main providers, such as Amazon EC2 or Windows Azure). Major public Cloud providers currently charge on a per-hour basis for a given computing capacity, regardless of their actual usage. If lower average memory consumption resulted in a lower cost, users could integrate these techniques in order to cut down costs, which would result in a win-win situation for both users and resource providers.

Concerning vertical elasticity, while open-source hypervisors such as KVM and Xen include support for techniques like memory ballooning, open source Virtual Infrastructure Managers (VIMs) such as OpenNebula and OpenStack do not currently include such support out of the box. In particular, this paper introduces support to vertical elasticity, through dynamic memory management, with a proof-of-concept implementation using an ad hoc modified version of OpenNebula [6] and the KVM hypervisor in order to dynamically shrink and grow the VMs' memory.

There are previous works in the area of elasticity on Cloud infrastructures. For example, the work by Ali-Eldin et al. [7] includes an adaptive horizontal elasticity controller for Cloud infrastructures, where a Cloud service is modelled by queue theory and service load is estimated to build proactive controllers. There are also works related to augmenting the computing capacities of mobile devices with the elastic capabilities of Cloud computing [8]. The aforementioned works focus on horizontal elasticity and do not address the topic of vertical elasticity, which is the main focus of this paper. Concerning vertical elasticity, the work by Kalyvianaki et al. [9], integrates the Kalman filter into feedback controllers to dynamically allocate CPU resources to VMs. The CPU utilization is tracked and the allocations are updated accordingly. The work by Zhao et al. [10] describes a system that monitors the memory usage of each VM to predict its memory needs and reallocate the host memory. They use the Xen hypervisor. In [11], Dawoud et al. focus on vertical elasticity and compare its benefits and drawbacks with respect to horizontal elasticity. They propose an Elastic VM architecture which scales number of cores, CPU capacity and memory, by using the Xen hypervisor. They show that by adapting the VM capacities to the requirements of the application (web-tier based), fine-grained resource provisioning is possible. Their case study exclusively focuses on dynamically altering the virtual CPUs and does not address memory scaling.

As opposed to previous works, this paper contributes a study of dynamic memory management on virtualized infrastructures for the execution of (scientific) applications with dynamic memory requirements. Techniques for vertical elasticity management are addressed and issues concerning the elasticity rules are covered, pointing out the main implications to be considered when deploying these techniques. A system has been implemented to support these techniques on our private Cloud infrastructure and a case study with different memory consumption patterns is executed in order to assess the effectiveness of an elastic management of the memory size of VMs.

After the introduction, the remainder of the paper is structured as follows. First of all, section 2 describes the methods employed to manage the vertical elasticity in terms of dynamic memory management. Then, section 3 describes the architecture employed to dynamically modify the memory size of the VMs, describing the proposed implementation. Next, section 4 describes a case study that executes a synthetic application that reproduces several dynamic memory consumption patterns on a virtual infrastructure. Later, section 5 discusses the main implications of the results both from the point of view of the user and the resource provider. Finally, section 6 summarises the paper and points to future work.



(a) Ideal situation (b) Real situation

Fig. 1. Memory provisioning approaches under ideal and real conditions.

2. Vertical Elasticity Management on Virtualized Environment

Virtualization provides the ability to simultaneously run multiple VMs on top of the same physical machine with the help of a hypervisor, which mediates access to physical hardware for the different VMs. The Operating Systems (OSs) running on the VMs are the so-called *guest OS*, and can be different from the *host OS*, the OS running on the host. In this paper we focus exclusively on GNU/Linux for both the host and guest OS, and the KVM hypervisor, which allows to change the capacities of a VM at runtime without service disruption.

Figure 1(a) summarises the main memory provisioning approaches employed when executing an application with dynamic memory requirements on a VM. As an example, it depicts an oscillatory memory pattern which resembles the one shown in the work by Pavlovic et al. [5] concerning the scientific application GADGET (GAlaxies with Dark matter and Gas intEracT). Since the user knows the memory requirements of the application, the most common approach is to overprovision the memory size of the VM on which to run the application. It is well known that an application that requests an amount of memory that cannot be kept as resident memory, requires swap space in order to supply the application with that amount of virtual memory. This can result in a phenomenon called *thrashing*, where the virtual memory subsystem is constantly paging, swapping pages from memory to disk and vice versa, thus causing the performance of the application to plummet. Therefore, overprovisioning memory aims at avoiding thrashing. However, in applications with dynamic memory requirements this results in a significant memory waste, since allocated memory to the VM is not always being used by the application. Consider the case of allocating resources from a pay-per-use Cloud provider that considered billing for memory usage. Overprovisioning would cause increased costs for a memory that is not always in use. On the flip side, it is clear that underprovisioning would cut down the costs at the expense of incurring in thrashing, which would seriously affect the level of performance of the application, something really undesired in the case of computationally intensive scientific applications.

Instead, elastic memory provisioning dynamically shrinks and grows the VM size in order to fit the memory consumption of the application. It is clear from Figure 1(a) that elastic provisioning has several advantages with respect to the previous aforementioned approaches. On the one hand, the VM dynamically requests an amount of memory large enough to consistently keep the application data in resident memory and consequently to preserve the performance level of the application. On the other hand, the unused memory is released by shrinking the VM memory size when the application no longer requires that amount of memory. This enables the host OS to take advantage of that spare memory.

Notice that Figure 1(a) represents ideal conditions, since resizing the VM before the application actually starts consuming more memory would require an *a priori* memory profile of the application in order to precisely adapt to the memory consumption changes before they actually occur. Instead, Figure 1(b) resumes the main problems one expects to find when trying to adapt the memory size of the VM to the application's memory consumption. The figure resumes the following scenario: VMs include an agent that periodically reports to an external monitor,

every few seconds, the free memory in the VM. The monitor decides to increase/decrease the memory size of the VM whenever a certain *elasticity rule* is triggered, which detects when a threshold has been surpassed.

Notice that in *A*, after the application has requested more memory than available (which typically succeeds if the VM has enough swap space), it is possible to increase the memory size of the VM up to a certain level (scale up). Depending on the rapid increment of memory of the application and the elasticity rule, the new memory size of the VM might not be enough to satisfy the requirements of the application (*B*). This would require another increment in the memory size of the VM. After the application starts releasing memory and the monitor detects that free memory is available, it decides to scale down to a certain amount of memory (*C*), which might require some adjustments (*D*) until the VM memory size can completely host the application in resident memory.

Therefore, adapting to the dynamic memory pattern of the application requires an accurate and frequent monitoring of the free memory. Besides, a proper design of elasticity rules is required to shrink and grow the memory size of the VM without incurring neither in excessive memory overprovisioning nor in thrashing.

3. Main Architecture and Implementation Details

This section describes the proposed architecture and implementation details of the solution developed to enable elastic memory management of VMs in order to automatically satisfy the dynamic memory requirements of applications.

It is responsibility of the hypervisor to support the techniques to enable vertical elasticity. In this work, we have focused on the open-source KVM hypervisor, which has achieved wide use and is broadly accepted in the virtualization community. As of early 2013, CPU Hotplug, i.e. the ability to increase the number of CPUs of a running VM is unsupported in KVM [12]. Fortunately, Memory Ballooning, i.e. the ability to increase/decrease the memory size of a running VM is fully supported in recent versions of the hypervisor. In addition, any recent Linux kernel supports this feature. This means that no kernel modification is required to take advantage of memory ballooning if using GNU/Linux as a guest OS. This technique is supported by the KVM `virtio_balloon` driver, which is basically a kernel driver inside the guest OS that behaves as a not-swappable process that can expand or shrink its memory usage, controlled by the hypervisor on the host OS. If the balloon expands, the physical memory available in the VM is reduced, that compels the guest OS to reduce the memory footprint of other processes when insufficient free memory is detected (for instance, by passing some of their memory pages to the swap space, or killing some of them in extreme situations). Then, the memory allocated by the balloon process in the guest OS can be reclaimed by the host OS, which can be possibly used by other VMs. Despite its complexity, this mechanism reacts almost instantaneously, that is, the guest OS reflects the memory change a few moments after the libVirt `setmem` operation is executed through `virsh`.

Currently, the main open-source Virtual Infrastructure Managers (VIM) such as OpenNebula and OpenStack do not support vertical elasticity out of the box. For this proof-of-concept implementation we have modified OpenNebula 3.8 in order to include an additional operation to access the functionality of the KVM hypervisor for memory ballooning, through the libVirt API. Therefore the new operation allows to increase and decrease the memory allocated to a given VM, from OpenNebula.

Figure 2 summarises the main components involved. First of all, the VMs are deployed via OpenNebula on a private Cloud infrastructure. They are based on a Virtual Machine Image (VMI) with Ubuntu 12.04 LTS, which has a kernel recent enough (3.2.X) to support memory ballooning techniques when running as a guest OS. The Ganglia Monitoring System is employed to periodically monitor the memory usage in each VM. The Ganglia monitoring daemon (GMond) is pre-installed in the VMI in order to report back to the Ganglia server the periodic state of the memory used of the VM (used memory, free memory and swap usage). The Vertical Elasticity Manager (VEM) includes a monitoring component based on the Ganglia Meta Daemon (gmetad), which aggregates the metrics obtained from the Ganglia agent running on the VM. The VEM also includes the implementation of the elasticity rules, which in our case aim at maintaining an user-defined percentage of free memory on the VM, called the Memory Overprovisioning Percentage (MOP). The idea is to avoid thrashing and, therefore, to keep the VM memory size beyond the limits of the resident memory used by the application. The elasticity rules enable the system to decide when to scale up or scale down, delegating on the Infrastructure Adapter which interacts with the VIM to effectively modify the VM memory size. The VEM can be seamlessly executed as part of the private

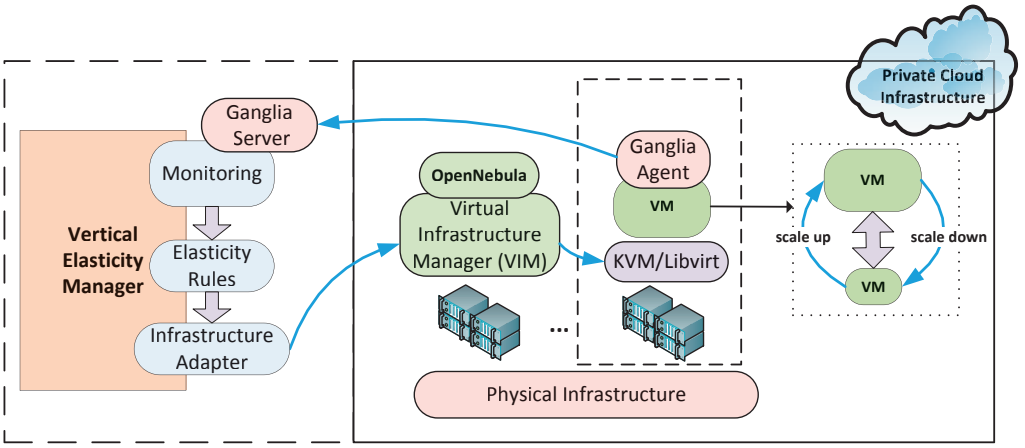


Fig. 2. Proposed implementation to enable elastic memory management.

Cloud infrastructure or as an external component to the Cloud, since it only requires inbound connectivity from the Ganglia Agent and outbound connectivity to the VIM. The VEM can simultaneously operate with several VMs to independently scale up and down different VMs, although this is not shown in the figure.

4. Case Study

In order to assess the effectiveness of these techniques, we have executed a case study on a private Cloud infrastructure composed by four Blade servers (M600 and M610 models). Each server has eight cores and 16 GB of RAM. VMs are deployed by OpenNebula and libVirt is employed to manage the memory ballooning techniques supported by the KVM hypervisor. VMs are based on Ubuntu 12.04 LTS with 1 GB of (initial) memory size and 1 GB of swap space. VMs run on top of KVM which in turn runs on top of a host OS (which again is Ubuntu 12.04). From the point of view of the host OS, separate VMs run as separate processes. Our hypothesis is that dynamically adjusting the memory size of a VM can be beneficial to the host OS, specially by rearranging the freed memory among other demanding tasks (or VMs running alongside). Therefore, it is important to estimate the reduction in the resident memory of the corresponding KVM process running on the host OS that the reduction in the VM memory size causes. The initial VM memory size is an upper bound that cannot be exceeded.

We have created a synthetic benchmark in order to reproduce dynamic memory consumption patterns of applications, such as those described in [5]. The program has an array allocated whose size is periodically adjusted (by the POSIX function `realloc`) in order to mimic a determinate memory consumption pattern. But memory allocations only modify the virtual memory requirements associated to the process and has no effect on its memory footprint on the physical memory. This is because modern operating systems actually allocate memory pages when they are accessed (by readings or writings), and use heuristics based on this utilization to assign the maximum resident memory of a process and to select which pages to move to the swap space when necessary. Therefore, the program tries to maintain the largest possible portion of the array in the physical memory by continuously performing a simple calculation that involves the whole array, $\sum_{i=0}^{n/2} a_{2i}\, a_{2i+1}$. We deliberately selected a calculation whose performance is limited by memory bandwidth (considering the current processors features), that requires two floating point operations (a multiplication and an addition, in single precision) for each two number reads from the memory. This pattern frequently appears in real applications like, for instance, scientific applications that perform operations with large arrays.

The Vertical Elasticity Manager (VEM) gathers the monitoring information every 5 seconds, supplied by the Ganglia monitoring daemon, in order to have an updated detail of the VM (or VMs) memory. Specifically, the used memory of a VM is estimated by subtracting the free, cached and buffers memory from the total memory,

whose values are reported on `/proc/meminfo` (these estimations also correspond to the used memory reported by the `free` command). We also prevent the VM from being shrank beyond 256 MB in order to leave room enough for the guest OS to properly operate. Even though the recommended minimum system requirements for Ubuntu 12.04 is 128 MB of RAM[1], we chose this limit after experiencing unexpected freezes in the VMs running below this memory limit. The elasticity rule is only applied if the percentage of the free memory of the VM is smaller than 80% or greater than 120% of the MOP. Under those circumstances, the VEM dynamically adapts the VM memory size by means of the following expression:

$$\text{VM memory size} = \text{used memory} \times (1 + \text{MOP})$$

This rule implies shrinking and growing the VM memory size depending on the behaviour of the application, and the magnitude of the memory variations depends on how fast or slow the application requests or releases memory. As an example, using a MOP of 10% means that the elasticity rule will only be triggered when the free memory of the VM is lower than 8% or greater than 12%. This enables the VEM to act only when substantial changes in the memory usage of the VM occur.

We considered two different values of MOP for the tests, 10% and 30%, responding to different approaches. A 10% value of MOP aims at reducing the unused memory of the VM, but has a higher chance to incur in thrashing if the application memory consumption grows faster than the rate at which the VEM increases the memory size. In contrast, a MOP of 30% aims at reducing the chance of thrashing if the memory consumption grows rapidly, but surely at the expense of wasting more memory.

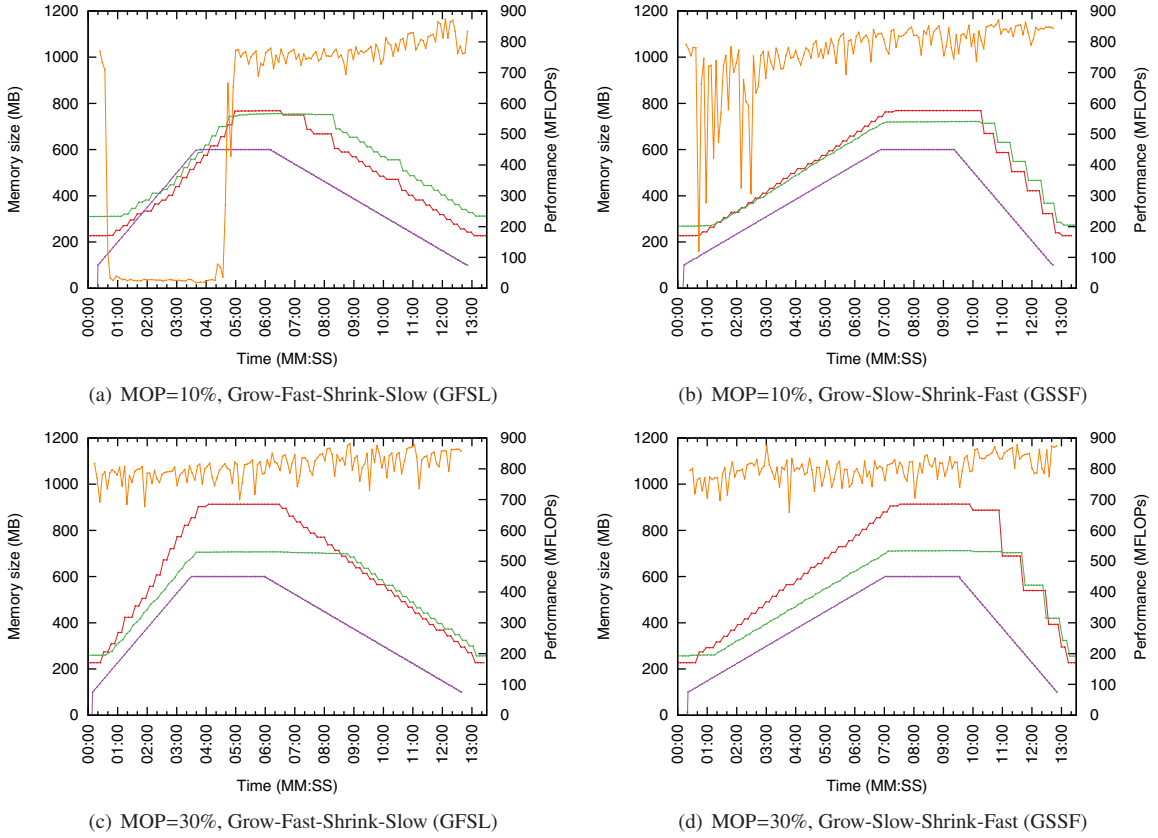
Two different patterns of application memory usage have been included. The first one describes an application which starts with an initial memory consumption and rapidly reaches a memory consumption plateau followed by a slower decline in memory requirements down to the initial level (Grow-Fast-Shrink-Slow, GFSL). The other pattern shows a slow increase in memory usage that is followed by a memory consumption plateau, before rapidly declining into the initial memory consumption (Grow-Slow-Shrink-Fast, GSSF). Having two different slopes in memory patterns enable to test different scenarios and cover a wider range of applications.

Figure 3 summarises the results obtained. Notice that four figures are shown, corresponding to testing two levels of MOP (10% and 30%) with two different memory consumption patterns (GFSL and GSSF). Figure 3(a) shows that when using a 10% MOP and an application that steeply grows in memory usage, the memory consumption grows faster enough for the VEM to be unable to properly adjust to the sudden and incremental memory usage. This means that the application is requiring more memory than it is available in the VM, what causes the application to start thrashing (to a virtual disk indeed) and performance (in terms of MFLOPs) to dramatically plummet. Notice that eventually the VEM achieves to increase the VM memory size up to the application requirements. At this point, the application performance is restored, because it stops thrashing. As soon as the application starts releasing memory, the adaptive controller accordingly reduces the memory of the VM, which in turn enables reducing the resident memory employed by the KVM process in the host OS. In fact, the resident memory of the KVM process is managed by the host OS just like any other process on the system. The average memory size of the VM was 495 MB and the average performance of the application was 490 MFLOPs.

In Figure 3(b), an extreme case occurs which is relevant for the discussion. The MOP is still 10% but now the application memory consumption grows at a slower pace than before. This means that the VEM is partially able to adjust to the incremental changes of memory requirements, so the application oscillates between thrashing and not, thus causing performance to drop at some points of the execution. Whenever the application stops thrashing to disk, performance is restored. For this case, the average memory size of the VM was 512 MB and the average performance was 767 MFLOPs.

Figures 3(c) and 3(d) show that increasing the MOP up to 30% enables the VEM to satisfy the dynamic memory requirements, under both memory consumption patterns. No performance loss is observed, and the ratio at which the VEM increments the VM memory size allows the application to be fully held in resident memory uninterruptedly. Notice that the drawback of a higher MOP is that more provisioned memory is wasted. For the case in Figure 3(c), the average memory size was 623 MB and the average performance was 807 MFLOPs, while in Figure 3(d), they were 627 MB and 802 MFLOPs respectively.

[1] https://help.ubuntu.com/12.04/serverguide/preparing-to-install.html



(a) MOP=10%, Grow-Fast-Shrink-Slow (GFSL)

(b) MOP=10%, Grow-Slow-Shrink-Fast (GSSF)

(c) MOP=30%, Grow-Fast-Shrink-Slow (GFSL)

(d) MOP=30%, Grow-Slow-Shrink-Fast (GSSF)

Fig. 3. Evolution of the VM memory size (—), the resident memory of the KVM process associated to the VM (—), the memory demanded by the application (—) and the application performance (—) for different configurations of Memory Overprovisioning Percentages (MOP) and memory consumption patterns.

In order to perform proper comparisons, the application was also executed on a similar VM with memory enough for the application to not incur in thrashing. We used a VM with 755 MB of memory size, since that was the maximum resident memory size of the corresponding KVM process when executing the application. This provides the perfect fit to cope with the maximum memory consumption of the application. In this VM, the application delivered an average performance of 805 MFLOPs. Therefore, it can be pointed out that thrashing caused a dramatic reduction on the performance of the application, as it is expected. Perhaps the most interesting situation is obtained with MOP 30%, regardless of the memory consumption pattern, where no performance loss is noticeable but a 17% reduction in memory size is achieved. In the case of using a MOP 10%, when the memory grows slowly (Figure 3(b)), a reduction of a 32% in memory size introduces a performance penalty of only a 4.65%. Notice that these memory reductions will be even larger in scenarios in which the VM memory size is clearly overdimensioned, since we used the minimum VM memory size that enables to fully host the application in memory without thrashing. Therefore, by adjusting the elasticity rules it is possible to alter the tradeoff between overprovisioning memory and reducing the chance to performance loss (due to thrashing).

5. Discussions

The results obtained by the previous case study pave the way for a thorough discussion. First of all, reducing the VM memory size causes a reduction on the corresponding KVM process running on the host OS. This is of paramount importance since, otherwise, the usage of these vertical elasticity techniques would pay no benefit from the perspective of the physical host. According to our tests, if the VM memory size remains constant and there

is free available memory on the host OS, a reduction of the memory consumption of the application (running on the guest OS) does not reduce the memory used by the corresponding KVM process. However, since VMs run as additional processes in the host OS, increasing the memory pressure (by exceeding the available RAM) on the host OS also reduces the resident memory used by each of the KVM processes.

Concerning the elasticity rules, it is important to properly adjust the Memory Overprovisioning Percentage (MOP) in order to cope with sudden increases in the memory requested by the application. Abrupt changes are more difficult to deal with than smoother slopes, since once the application starts swapping excessively to disk, the main indicator on which our elasticity rules rely (free memory of the VM) is no longer valid. In that case, it is crucial to rapidly increase the memory size of the VM in order to stop the application from thrashing and restore its level of performance. This means that elasticity rules should be accompanied by fail-safe mechanisms that are triggered when the application is detected to be thrashing and which rapidly increase the VM memory size. For that, an agent could be deployed in the VM to monitor not only the memory usage of the VM (in our current approach), but also the application memory usage. This way, by having more accurate information it is possible to better cope with the sudden memory changes. In addition, notice that there can be other approaches when for applying the elasticity rules. For example, the monitoring information could be averaged among several time periods in order to better cope with abrupt oscillations in the memory consumption. These changes might provide a better adaptation, and we plan to integrate them in future works.

Public Cloud providers such as Amazon EC2 currently offer a pay-per-usage on which, for example, an *m1.small* instance (VM) deployed in the US East region costs 0.065$ per hour, and features 1.7 GB of memory. If an application has an average memory demanding of 1 GB but with short peak memory demanding instants of 5 GB, an *m1.large* instance which offers 7.5 GB is required instead, at 0.260$ per hour. That price is regardless the actual usage in terms of CPU and memory by the instance. As technology progresses, users expect more competitive prices, and that could come from the public Cloud providers introducing these vertical elasticity techniques for dynamic memory management. This would also facilitate the users to choose an instance type, due to the reduction in the penalty cost of overestimating the requirements of an application. Moreover, apart from users' and providers' economic benefits, these techniques can effectively reduce thrashing, which not only drops dramatically the performance of applications but also wastes CPU time and disk transference and finally, energy.

Notice that these techniques could also be employed directly by the Cloud provider without even involving the user. By proper monitoring and dynamic memory management, the provider could dynamically change the VM memory size at runtime, without affecting the performance of the application. As a result, more VMs could be run per physical host, thus enhancing server utilization and achieving oversubscription, where the provider expects that aggregate VM resource demands at any point do not exceed the resources of the physical machine. Under this scenario, coping with sudden overloads might require techniques such as VM migration or network memory, as described in the work by Williams et al. [13].

As an additional comment, the dynamic memory management of VMs is also of interest to VIMs, such as OpenNebula or OpenStack, in order to enhance resource utilization. The ability to dynamically change the VM memory size, either specified by the user, or by the VIM itself by proper monitoring of resource consumption, enables to better decide a proper allocation of VMs to physical hosts. Many VIMs provide static allocations of resources to VMs, regardless of the actual usage of those resources. For example, only four VMs of 1 GB size would be allocated to a physical host with 4 GBs of memory, independently of the actual memory consumption of those VMs. By leveraging dynamic size scaling, the VIM would be able to oversubscribe physical resources and rely on the aforementioned migration techniques should the memory requirements of the VM exceed the physical resources available. In scenarios in which users typically overprovision their memory requirements, these techniques could certainly pay off for a better resource utilization.

6. Conclusions and Future Work

This paper has described the main implications of integrating vertical elasticity techniques on the execution of applications with dynamic memory requirements in a virtualized environment. In particular, dynamic memory management has enabled to fit the Virtual Machine size to the changing memory consumption of the application at runtime. A system has been developed to monitor the VM memory and apply vertical elasticity rules in order to dynamically change its memory size by using the memory ballooning technique provided the KVM hypervisor.

The results have shown that it is possible to adapt the VM memory size in order to reduce the memory consumption of the VM while maintaining the level of performance of the running application. This adjustment is beneficial for the resource provider, since the host can take advantage of that memory reduction in order to dedicate it for other processes (possibly other concurrent VMs). In a future scenario, the progress of technology might enable public Cloud providers to charge for memory consumption on a per-minute level. This way, these techniques would enable a significant gain for both the user, whose application might still run at peak performance, and the provider, who can better benefit from oversubscribing the physical resources.

Notice that a relatively simple elasticity rule has been used to adapt the VM memory size to the application requirements. More advanced rules could also be employed, such as using a dynamic MOP or trying to forecast the application memory requirements depending on its previous memory consumption patterns. In addition, an alternative and more efficient approach for deterministic applications could be to generate a runtime memory consumption profile of the application by executing it at the target destination. This profile would be used as input to the Vertical Elasticity Manager in order to precisely increase the VM memory size right before the actual memory increase of the application takes place. This would disable the elasticity rules and, instead of a reactive behaviour, where actions are taken when triggers are reached, a proactive behaviour would be exposed, thus obtaining a better fit of the VM memory size to the application's memory consumption.

Finally, if thrashing wants to be avoided at all, it could be of interest to explore techniques available in current debuggers in order to attach to a process and detect when a memory increase operation is performed. This could temporarily hold the application while the Vertical Elasticity Manager is notified of the request to increase the memory requirements and proceeds with the VM memory adjustment. The application would resume execution when the underlying VM has the appropriate memory size. Although this could be done in the order of a second, it would be important to quantify the performance penalties of this approach, which will be explored in future works.

Acknowledgements

The authors would like to thank the financial support received from the Generalitat Valenciana for the project GV/2012/076 and to the Ministerio de Economía y Competitividad for the project CodeCloud (TIN2010-17804). The authors would also like to thank David Lozano for his fruitful comments and implementation support.

References

[1] P. Mell, T. Grance, The NIST Definition of Cloud Computing. NIST Special Publication 800-145 (Final), Tech. rep. (2011).
URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[2] M. Armbrust, A. Fox, R. Griffith, A. Joseph, Above the clouds: A berkeley view of cloud computing, Tech. rep., UC Berkeley Reliable Adaptive Distributed Systems Laboratory (2009).
URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.149.7163\&rep=rep1\&type=pdf

[3] E. Elmroth, J. Tordsson, F. Hernández, Self-management challenges for multi-cloud architectures, Towards a Service-Based Internet. Lecture Notes in Computer Science 6994 (2011) 38–49.
URL http://www.springerlink.com/index/KP83561G0433J632.pdf

[4] E. G. Report, The Future of Cloud Computing. Opportunities for European Cloud Computing Beyond 2010, Tech. rep., European Comission (2010).

[5] M. Pavlovic, Y. Etsion, A. Ramirez, On the memory system requirements of future scientific applications: Four case-studies, in: 2011 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2011, pp. 159–170. doi:10.1109/IISWC.2011.6114176.
URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6114176

[6] B. Sotomayor, R. S. Montero, I. M. Llorente, I. Foster, Virtual infrastructure management in private and hybrid clouds, IEEE Internet Computing 13 (5) (2009) 14–22. doi:10.1109/MIC.2009.119.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5233608

[7] A. Ali-Eldin, J. Tordsson, E. Elmroth, An adaptive hybrid elasticity controller for cloud infrastructures, in: 2012 IEEE Network Operations and Management Symposium, no. 978, Ieee, 2012, pp. 204–212. doi:10.1109/NOMS.2012.6211900.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6211900

[8] X. Zhang, A. Kunjithapatham, S. Jeong, S. Gibbs, Towards an Elastic Application Model for Augmenting the Computing Capabilities of Mobile Devices with Cloud Computing, Mobile Networks and Applications 16 (3) (2011) 270–284. doi:10.1007/s11036-011-0305-7.
URL http://www.springerlink.com/index/10.1007/s11036-011-0305-7

[9] E. Kalyvianaki, T. Charalambous, S. Hand, Self-adaptive and self-configured CPU resource provisioning for virtualized servers using Kalman filters, in: Proceedings of the 6th international conference on Autonomic computing - ICAC '09, ACM Press, New York, New

York, USA, 2009, p. 117. doi:10.1145/1555228.1555261.
URL http://dl.acm.org/citation.cfm?id=1555228.1555261

[10] W. Zhao, Z. Wang, Y. Luo, Dynamic memory balancing for virtual machines, ACM SIGOPS Operating Systems Review 43 (3) (2009) 37. doi:10.1145/1618525.1618530.
URL http://dl.acm.org/citation.cfm?id=1618525.1618530

[11] W. Dawoud, I. Takouna, C. Meinel, Elastic Virtual Machine for Fine-Grained Cloud Resource Provisioning, in: Global Trends in Computing and Communication Systems, 2012, pp. 11–25.
URL http://www.springerlink.com/index/P663M162G7V2L1T5.pdf

[12] KVM, KVM CPU Hotplug.
URL http://www.linux-kvm.org/page/CPUHotPlug

[13] D. Williams, H. Jamjoom, Y.-H. Liu, H. Weatherspoon, Overdriver: handling memory overload in an oversubscribed cloud, ACM SIGPLAN Notices 46 (7) (2011) 205. doi:10.1145/2007477.1952709.
URL http://dl.acm.org/citation.cfm?id=2007477.1952709